Docket: 43876-156 **PATENT**

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Application of	:	Customer Number: 20277
HANSEN et al.	:	Confirmation Number: 5955
Application No.: 10/757,866	:	Group Art Unit: 2183
Filed: January 16, 2004	:	Examiner: Eric Coleman

For: METHOD AND SOFTWARE FOR STORE MULTIPLEX OPERATION

SECOND SUPPLEMENTAL DECLARATION OF KORBIN VAN DYKE

I, Korbin Van Dyke, state that:

Summary of My Opinions

1. I previously submitted two declarations in connection with this proceeding (see Van Dyke Declaration dated August 15, 2007, and Van Dyke declaration dated May 2, 2008, referenced hereinafter as the "initial Van Dyke declaration" and the "supplemental Van Dyke declaration". For brevity, I will not repeat information set forth in the initial or supplemental Van Dyke declarations in this declaration.

2. In preparation of this declaration I have reviewed U.S. Patent Application Serial No. 10/757,866. I have also reviewed U.S. Patent Nos. 5,742,840 and 6,295,599 (respectively the '840 and '599 patents) that the 10/757,866 patent application indirectly claims priority to, as well as appendices to the '840 and '599 patents (the Terpsichore and Zeus System Architecture manuals, respectively, and hereinafter referred to respectively as the Terpsichore and the Zeus manuals). I have reviewed the Office Action for the 10/757,866 patent application mailed on July 16, 2008, including the paragraphs on pages 2-3 that discuss the Examiner's rejection of claims 1-18 and 28-41 for failing to comply with the written description requirement of § 112, and the paragraphs on pages 3-4 that discuss the Examiner's rejection of claims 1-18 and 28-41 for failing to comply with the enablement requirement of § 112. I have also reviewed the paragraph on page 15 that discusses the Response to Arguments and particularly the Examiner's statement that the "instructions cited in the remarks, are deemed as performing an operation that is set and the bits that are masked are predetermined and which bits are to be enabled or disabled

for write are predetermined and set" and the "Applicant cited instructions in '599 and '840 patents do not provide for individually selecting a bit of an operand of and operand for enabling or disabling write of the bits during processing of the instruction".

3. My understanding is that the features of the claimed invention are taught and supported by complying with the written description requirement and the enablement requirement. My understanding of the written description requirement is that a patent disclosure must describe the claimed invention in sufficient detail that one of ordinary skill in the art can reasonably conclude that the inventor had possession of the claimed invention at the time of filing the patent disclosure. My understanding of the enablement requirement is that the patent disclosure must contain sufficient information regarding the subject matter of the claims to enable one of ordinary skill in the pertinent art to make and use the claimed invention. I further understand that whether the enablement requirement is met depends on whether undue experimentation is necessary for one of skill in the art to practice the invention in light of the patent disclosure.

4. Based on my review of the materials identified in paragraph 2 of this declaration, it is my opinion that with respect to the following limitations relating to (amended) claims 1, 10, 28, and 35, the disclosures of the '840 patent and the '599 patent each indicate that the inventors were in possession of the claimed invention of the 10/757,866 patent application as of the August 16, 1995 filing date of the '840 patent and further as of the August 24, 1999 filing date of the '599 patent; and further the disclosures of the '840 patent and the '599 patent each would have enabled a person of ordinary skill in the art to make and use, without undue experimentation, the claimed invention of the 10/757,866 patent application as of the August 16, 1995 filing date of the '840 patent, and further as of the August 24, 1999 filing date of the '599 patent. The limitations referred to are:

{claims 1 and 10} "the mask consisting of N independently selectable mask bits, N being an integer multiple of eight, each of the mask bits corresponding to a data bit contained in the at least one register, each of the mask bits being independently selectable as either a write-enabled mask bit or a write-disabled mask bit"

{claim 28} "the second operand consisting of N independently selectable bits, N being an integer multiple of eight, wherein each bit in the second operand is independently selectable as either having a first predetermined value or a second predetermined value; and for each bit in the first operand, the bitwise insert operation inserting the bit into a corresponding bit position in a destination value if a corresponding bit in the second operand has the first predetermined value"

{claim 35} "the second operand consisting of N independently selectable bits, N being an integer multiple of eight, wherein each bit in the second operand is independently selectable as either having a first predetermined value or a second predetermined value; and wherein for each bit in the first operand, the bitwise insert operation inserts the bit into a corresponding bit position in a destination value if a corresponding bit in the second operand has the first predetermined value"

Summary of '840 Analysis:

5. The disclosure of the '840 patent provides detailed information and description that I believe indicates that the inventors were in possession of the aforementioned limitations of (amended) claims 1 and 10 of the 10/757,866 patent application, and that I further believe would have enabled a person of ordinary skill in the art to make and use the claimed invention without undue experimentation. For example, on at least pages 24-25 (describing general registers), 26 (generally describing store instructions), and 150-157 of the Terpsichore manual (describing details of Store and Store Immediate instructions such as various forms of Store Immediate and Store Multiplex Immediate instructions) there are detailed descriptions of the aforementioned claim elements.

6. The disclosure of the '840 patent provides detailed information and description that I believe indicates that the inventors were in possession of the aforementioned limitations of (amended) claim 28 and the substantially similar aforementioned limitations of (amended) claim 35 of the 10/757,866 patent application, and that I further believe would have enabled a person of ordinary skill in the art to make and use the claimed invention without undue experimentation. For example, on at least pages 24-25 (describing general registers), 26 (generally describing store instructions), and 150-157 of the Terpsichore manual (describing details of Store and Store Immediate instructions such as various forms of Store Immediate and Store Multiplex Immediate instructions) there are detailed descriptions of the aforementioned claim elements.

Summary of '599 Analysis:

7. The disclosure of the '599 patent provides detailed information and description that I believe indicates that the inventors were in possession of the aforementioned limitations of (amended) claims 1 and 10 of the 10/757,866 patent application, and that I further believe would have enabled a person of ordinary skill in the art to make and use the claimed invention without undue experimentation. For example, on at least pages 19-20 (describing general registers), 21 (generally describing store instructions), 123-125, and 128-130 of the Zeus manual (describing details of Store and Store Immediate instructions, including Store Multiplex and Store Multiplex Immediate forms) there are detailed descriptions of the aforementioned claim elements.

8. The disclosure of the '599 patent provides detailed information and description that I believe indicates that the inventors were in possession of the aforementioned limitations of (amended) claim 28 and the substantially similar aforementioned limitations of (amended) claim 35 of the 10/757,866 patent application, and that I further believe would have enabled a person of ordinary skill in the art to make and use the claimed invention without undue experimentation. For example, on at least pages 19-20 (describing general registers), 21 (generally describing store instructions), 123-125, and 128-130 of the Zeus manual (describing details of Store and Store Immediate instructions, including Store Multiplex and Store Multiplex Immediate forms) there are detailed descriptions of the aforementioned claim elements.

9. A detailed explanation of the basis for my opinions is set forth in the remainder of this declaration.

Detailed Basis for My Opinions

Analysis of the disclosures of the '840 and the '599 patents:

10. For brevity, the following analysis focuses on and provides details relating to the '840 patent, while reciting summary information pointing out where similar descriptive information is provided in the '599 patent.

11. As discussed in paragraph 20 of the initial Van Dyke declaration, the '840 patent describes structure of a general purpose, programmable media processor (including, for example, a register file and an execution unit), and the '840 patent recites that an instruction set for the general purpose media processor is described by the Microfiche Appendix (referred to herein as the Terpsichore manual). Similarly, the '599 patent describes structure of a general purpose, programmable processor for broadband applications, and the '599 patent includes and refers to a Microfiche Appendix (referred to herein as the Zeus manual) that describes, for example, an instruction set for the general purpose processor.

12. The Terpsichore manual of the '840 patent describes all of the aforementioned limitations of (amended) claims 1, 10, 28, and 35, on at least pages 150-157 (included in attached Exhibit A), with additional information on page 24 (also included in the attached Exhibit A). Paragraphs 22-28 of the initial Van Dyke declaration discuss aspects of those pages in detail. Paragraphs 13-30 of this declaration discuss particular aspects of those pages with respect to the aforementioned elements of (amended) claims 1, 10, 28, and 35. The Zeus manual of the '599 patent describes all of the aforementioned limitations of (amended) claims 1, 10, 28, and 35 on at least pages 19-20, 123-125, and 128-130 (attached as Exhibit B).

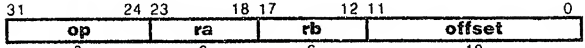
13. The Terpsichore manual, for example on pages 150-157, describes several variations of Store and Store Immediate instructions, including several Store Multiplex forms ("S.MUX.64.B.A" and "S.MUX.64.L.A") and several Store Multiplex Immediate forms ("S.MUX.64.B.A.I" and "S.MUX.64.L.A.I"):

S.MUX.64.B.A	Store multiplex octlet big-endian aligned
S.MUX.64.L.A	Store multiplex octlet little-endian aligned

S.MUX.64.B.A.I	Store multiplex octlet big-endian aligned immediate
S.MUX.64.L.A.I	Store multiplex octlet little-endian aligned immediate

14. The following discussion focuses on the Store Multiplex Immediate forms. As will be subsequently described, the Store Multiplex forms are similar to the Store Multiplex Immediate forms with respect to the claimed "N independently selectable mask bits" (amended claims 1 and 10) and "N independently selectable bits" (amended claims 28 and 35).

15. The Terpsichore manual, on page 155, describes an instruction format for the Store Immediate instructions:



As evidenced by the instruction format, the operands of the Store Immediate Instruction are '*ra*', '*rb*', and '*offset*'.

16. The Terpsichore manual, on pages 155-157, describes operation of the Store Multiplex Immediate form of the Store Immediate Instructions. As described, the operation includes:

(a) Determining '*function*' as '*MUX*';

SMUX64BAI, SMUX64LAI:
function ← MUX

(b) Determining '*size*' as '*64*';

S64LI, S64LAI, S64BI, S64BAI, SAAS64BAI, SAAS64LAI,
SCAS64BAI, SCAS64LAI, SMAS64BAI, SMAS64LAI, SMUX64BAI, SMUX64LAI
size ← 64

(c) Determining '*rsize*' as '*128*',

SCAS64BAI, SCAS64LAI, SMAS64BAI, SMAS64LAI, SMUX64BAI, SMUX64LAI:
rsize ← 128

(d) Reading a 64-bit value from a register specified by the operand '*ra*', and determining a virtual address based on the value from the register and the operand '*offset*' from the instruction;

a ← RegRead(ra, 64)
VirtAddr ← a + ($offset_{11}^{50}$ || offset)

(e) Reading a 128-bit value (based on '*rsize*' being 128) from a pair of registers specified by the operand '*rb*', and setting '*m*' to the value read from the register pair. The most-significant 64 bits of '*m*' are set to the value of the odd register of the pair and the least-significant 64 bits of '*m*' are set to the value of the even register of the pair;

m ← RegRead(rb, rsize)

(f) Reading a 64-bit value (based on '*size*' being 64) from memory from a location specified by the virtual address, and setting '*b*' to the value read;

Computing a new value to store to memory, '*n*', by logically bit-wise ANDing the most-significant 64 bits of '*m*' with the least-significant 64 bits of '*n*', logically bit-wise ANDing the 64-bit value read from memory, '*b*', with a logical bit-wise NOT of the least-significant 64 bits of '*m*', and logically bit-wise ORing the results of the two ANDing operations; and

Storing '*n*' as a 64-bit value (based on '`size`' being 64) to memory to the location specified by the virtual address.

MUX:
b ← LoadMemory(VirtAddr,size,order)
n ← ($m_{127..64}$ & $m_{63..0}$) | (b & ~$m_{63..0}$)
StoreMemory(VirtAddr,size,order,n)

Where, as understood by one of ordinary skill in the art:
'$m_{127..64}$' signifies the most-significant 64 bits of '*m*',
'$m_{63..0}$' signifies the least-significant 64 bits of '*m*',
'~' signifies a bit-wise logical NOT
'&' signifies a bit-wise logical AND
'|' signifies a bit-wise logical OR

17. The operations described in paragraphs 16(e) and 16(f) (above) result in selectively replacing bits in the memory location (at the virtual address as specified by the '`ra`' and '`offset`' operands) with bits from the odd register (of the register pair specified by the '`rb`' operand). Which of the selectively replaced bits are replaced is determined by the value from the even register (of the register pair specified by the '`rb`' operand).

18. One of ordinary skill in the art would understand the following specific examples of operation of the Store Multiplex Immediate instruction. Note that underscores ('_') are used to improve readability, and have no substantive meaning.

(a) Execution of: `S.MUX.64.B.A.I(operand rb = 24)`
With starting values of:

```
Mem[VA] = 01010101_00000000_11111111_10101010
REG[24] = 00000000_01010101_10101010_00000000 # Mask
REG[25] = 11111111_11111111_00000000_00000000 # Data
```

Results in:

```
Mem[VA] = 01010101_01010101_01010101_10101010
```

(b) Execution of: `S.MUX.64.B.A.I(operand rb = 24)`
With starting values of:

```
Mem[VA] = 01010101_00000000_11111111_10101010
REG[24] = 00000000_00010001_10111010_00000000 # Mask
REG[25] = 11111111_11111111_00000000_00000000 # Data
```

Results in:

```
Mem[VA] = 01010101_00010001_01000101_10101010
```

19. The example illustrated in paragraph 18(a) (above) illustrates a resultant value in memory at a particular virtual address (Mem[VA]). The resultant memory value is determined from a starting value from memory at the particular virtual address and values in a register pair specified by the rb operand of the instruction. The even register of the register pair, REG[24], functions as a mask, and the odd register of the pair, REG[25], functions as data to replace, according to the mask, selected bits of memory at the particular virtual address. For every bit that is asserted in the even register (REG[24]), a corresponding respective bit is selected from the odd register (REG[25]) to replace a corresponding respective bit in memory.

20. The example illustrated in paragraph 18(b) (above) is identical to that illustrated in paragraph 18(a), except that for illustration, three bits of the mask (the value in the even register of the register specified by the rb operand) are logically inverted values, and consequently corresponding respective bits of the result in memory are different than in the paragraph 18(a) example (see red box, bold, and underlining highlighting). In general, there are no restrictions on possible bit patterns for the mask; any of 2^n unique mask patterns are possible (where 'n' is the width, in bits, of the mask). Thus the least-significant bit of the mask may selectively be either one or zero, in combination with the next-to-least-significant bit of the mask selectively being either one or zero, and so forth, up to and including the most-significant bit of the mask. The value of any bit in the mask is not related to the value of any other bit in the mask; the values of the bits of the mask are independent from one another.

21. The operation of the Store Multiplex forms are similar to the Store Multiplex Immediate forms, except the virtual address is computed by summing values from two registers specified by the '*ra*' and '*rb*' operands, and the register pair (with the mask and the data) is specified by the '*rc*' operand.

Claims 1 and 10 (as amended)

22. The aforementioned limitations of (amended) claims 1 and 10 are:

(a) the mask consisting of N independently selectable mask bits,

(b) N being an integer multiple of eight,

(c) each of the mask bits corresponding to a data bit contained in the at least one register,

(d) each of the mask bits being independently selectable as either a write-enabled mask bit or a write-disabled mask bit

As is discussed by following paragraphs 23-25, all of the aforementioned claim limitations are described in the Terpsichore manual, at least on pages 24 and 150-157 (also see the discussion of those pages in paragraphs 13-21 of this declaration).

23. The element the mask consisting of N independently selectable mask bits is described by the Terpsichore manual, and corresponds, for example, to the even register of the register pair specified by the '*rb*' operand of the Store Multiplex Immediate instruction. The bits are independently selectable in that any bit is selectable as a zero or a one, irrespective of values of any of the other bits. The element N being an integer multiple of eight is also described by the Terpsichore manual, where N corresponds, for example, to 64, since the even register consists of 64 bits, and 64 is an integer multiple of eight.

24. The element each of the mask bits corresponding to a data bit contained in the at least one register is described by the Terpsichore manual, as illustrated for example by the description of the computation of the new value to store in memory. The computation recites (in part) "$m_{127..64}$ & $m_{63..0}$", where $m_{127..64}$ is the odd register of the register pair specified by the '*rb*' operand of the Store Multiplex Immediate instruction, and $m_{63..0}$ is the low register of the register pair. The '&' denotes a bit-wise logical AND operation, $m_{127..64}$ and $m_{63..0}$ are each 64 bits, and thus there is a bit in the mask for every bit in the data.

25. The element each of the mask bits being independently selectable as either a write-enabled mask bit or a write-disabled mask bit is described by the Terpsichore manual. The claimed "mask" corresponds to the even register of the register pair specified by the '*rb*' operand of the Store Multiplex Immediate instruction. Every bit of the even register is selectable as either a zero or a one, without regard to values of other bits of the even register. For every bit of the even register that is a one (corresponding to the claimed "write-enabled mask bit"), a corresponding bit is replaced in a memory location with a corresponding bit from the odd register of the specified register pair. For every bit of the even register that is a zero (corresponding to the claimed "write-disabled mask bit"), a corresponding bit in the memory location is left unchanged. The replacement of the memory location bit(s) is via a read of the memory location, a computation of a new memory value, and a write of the new memory value to the memory location, recited respectively in the Terpsichore manual as:

b ← `LoadMemory(VirtAddr, size, order);`
n ← $m_{127..64}$ & $m_{63..0}$) | (b & ~ $m_{63..0}$); and
`StoreMemory(VirtAddr, size, order, n).`

26. Thus all of the aforementioned limitations of (amended) claims 1 and 10 are described by the '840 patent, at least in pages 24 and 150-157 of the Terpsichore manual, and are also described by the '599 patent, at last in pages 19-20, 123-125, and 128-130 of the Zeus manual.

Claims 28 and 35 (as amended)

27. The aforementioned limitations of (amended) claims 28 and 35 are:

(a) the second operand consisting of N independently selectable bits, N being an integer multiple of eight,

(b) wherein each bit in the second operand is independently selectable as either having a first predetermined value or a second predetermined value,

(c) and (wherein) for each bit in the first operand, the bitwise insert operation inserting (inserts) the bit into a corresponding bit position in a destination value if a corresponding bit in the second operand has the first predetermined value

As discussed by following paragraphs 28-29, all of the aforementioned claim limitations are described in the Terpsichore manual, at least on pages 24 and 150-157 (also see the discussion of those pages in paragraphs 13-21 of this declaration).

28. The elements the second operand consisting of N independently selectable bits and wherein each bit in the second operand is independently selectable as either having a first predetermined value or a second predetermined value are described by the Terpsichore manual. The claimed "second operand" corresponds, for example, to the even register of the register pair specified by the '*rb*' operand of the Store Multiplex Immediate instruction. The bits are independently selectable in that any bit is selectable as a zero (the claimed "second predetermined value") or a one (the claimed "first predetermined value"), irrespective of values of any of the other bits. The element N being an integer multiple of eight is also described by the Terpsichore manual, where N corresponds, for example, to 64, since the even register consists of 64 bits, and 64 is an integer multiple of eight.

29. The element (wherein) for each bit in the first operand, the bitwise insert operation inserting (inserts) the bit into a corresponding bit position in a destination value if a corresponding bit in the second operand has the first predetermined value is described by the Terpsichore manual. The claimed "first operand" corresponds, for example, to the odd register of the register pair specified by the '*rb*' operand of the Store Multiplex Immediate instruction, and the claimed "second operand" corresponds to the even register of the register pair. For every bit of the even register that is a one (corresponding to the claimed "first predetermined value"), a corresponding bit is replaced in a memory location (corresponding to the claimed "destination value") with a corresponding bit from the odd register. For every bit of the even register that is a zero (corresponding to the claimed "second predetermined value"), a corresponding bit in the memory location is left unchanged. The replacement of the memory location bit(s) is via a read of the memory location, a computation of a new memory value, and a write of the new memory value to the memory location, recited respectively in the Terpsichore manual as:

b ← `LoadMemory(VirtAddr, size, order);`
n ← $m_{127..64}$ & $m_{63..0}$) | (b & ~ $m_{63..0}$); and
`StoreMemory(VirtAddr, size, order, n).`

30. Thus all of the aforementioned limitations of (amended) claims 28 and 35 are described by the '840 patent, at least in pages 24 and 150-157 of the Terpsichore manual, and are also described by the '599 patent, at last in pages 19-20, 123-125, and 128-130 of the Zeus manual.

Summary and Closing:

31. The '840 patent, including the Terpsichore manual, provides sufficient information in sufficient detail describing the claimed invention (as amended) of the 10/757,866 patent application, that one of ordinary skill in the art would reasonably conclude that the inventors had possession of the claimed invention at the time of filing the '840 patent. Further, the '840 patent, including the Terpsichore manual, provides sufficient information regarding the subject matter of the claimed invention (as amended) of the 10/757,866 patent application to enable one of ordinary skill in the pertinent art to make and use the claimed invention without undue experimentation. In addition, the '599 patent, including the Zeus manual, provides sufficient information in sufficient detail describing the claimed invention (as amended) of the 10/757,866 patent application, that one of ordinary skill in the art would reasonably conclude that the inventors had possession of the claimed invention at the time of filing the '599 patent. Further, the '599 patent, including the Zeus manual, provides sufficient information regarding the subject matter of the claimed invention (as amended) of the 10/757,866 patent application to enable one of ordinary skill in the pertinent art to make and use the claimed invention without undue experimentation.

32. Therefore, I believe that each of the '840 patent, including the Terpsichore manual, and the '599 patent, including the Zeus manual, provide adequate written description and enablement as required by 35 USC § 112 for the aforementioned limitations of (amended) claims 1, 10, 28, and 35 of the 10/757,866 patent application, as discussed in paragraph 4 of this declaration.

33. I have had no communication with any of the inventors of the 10/757,866 patent application (Craig Hansen and John Moussouris) relating to any material in this declaration.

34. I have been hired as a consultant in connection with procedures before the United States Patent and Trademark Office (USPTO) regarding patents and patent applications assigned to Microunity Systems Engineering, Inc., including the media processor patent application. I am being compensated for my services at the rate of $375/hour. Other than acting as a consultant in connection with procedures before the USPTO, I have no interest or connection with Microunity Systems Engineering, Inc.

35. During my evaluation of the media processor patent application, I have been impressed by the thoroughness and overall high-quality of the Terpsichore and Zeus manuals. The manuals provide clear and unambiguous descriptions of media processing systems and are thorough and well-written. The manuals provide comprehensive descriptions of instructions in complete architectural detail. The information in the manuals would have been readily understood and easily accessible to software engineers coding the media processing systems, and

hardware engineers implementing microprocessors for use in the media processing systems, and that is exactly what architecture reference manuals should be. This is not surprising, since the '840 patent and the '599 patent each include an architecture manual that is intended to enable hardware engineers to do exactly that – design, build, and implement a media processor that would include circuitry for the claim limitations set forth in paragraph 4 of this declaration, as described in the Terpsichore and the Zeus architecture manuals.

36. I hereby declare that all statements made herein are of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issuing therefrom.

Date: 2008 Oct 14

Korbin Van Dyke: K. Van Dyke

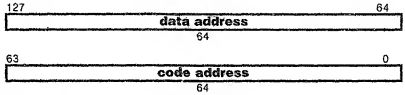
Address: 3343 Little Valley Rd.
Sunol, CA 94586

EXHIBIT A

Second Supplemental Declaration of Korbin Van Dyke

Pages 24 and 150-157 of Terpsichore System Architecture manual
(from Microfiche Appendix of the '840 patent)

The gateway contains two data items within its structure, a code address and a data address:

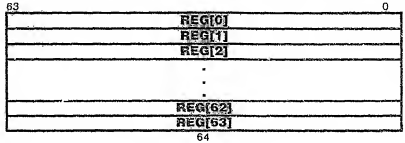


User state

The user state consists of hardware data structures that are accessible to all conventional compiled code. The Terpsichore user state is designed to be as regular as possible, and consists only of the general registers, the program counter, and virtual memory. There are no specialized registers for condition codes, operating modes, rounding modes, integer multiply/divide, or floating-point values.

General Registers

Terpsichore user state includes 64 general registers. All are identical; there is no dedicated zero-valued register, and there are no dedicated floating-point registers.



Definition

```
def val ← RegRead(m. size)
    case size of
        64:
            val ← REG[m]
        128:
            if m0 then
                raise ReservedInstruction
            endif
            val ← REG[m+1] || REG[m]
    endcase
enddef
```

Store

These operations add the contents of two registers to produce a virtual address, and store the contents of a register into memory.

Operation codes

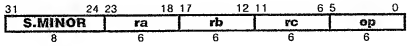
S.8[46]	Store byte
S.16.B	Store double big-endian
S.16.B.A	Store double big-endian aligned
S.16.L	Store double little-endian
S.16.L.A	Store double little-endian aligned
S.32.B	Store quadlet big-endian
S.32.B.A	Store quadlet big-endian aligned
S.32.L	Store quadlet little-endian
S.32.L.A	Store quadlet little-endian aligned
S.64.B	Store octlet big-endian
S.64.B.A	Store octlet big-endian aligned
S.64.L	Store octlet little-endian
S.64.L.A	Store octlet little-endian aligned
S.128.B	Store hexlet big-endian
S.128.B.A	Store hexlet big-endian aligned
S.128.L	Store hexlet little-endian
S.128.L.A	Store hexlet little-endian aligned
S.AAS.64.B.A	Store add-and-swap octlet big-endian aligned
S.AAS.64.L.A	Store add-and-swap octlet little-endian aligned
S.CAS.64.B.A	Store compare-and-swap octlet big-endian aligned
S.CAS.64.L.A	Store compare-and-swap octlet little-endian aligned
S.MAS.64.B.A	Store multiplex-and-swap octlet big-endian aligned
S.MAS.64.L.A	Store multiplex-and-swap octlet little-endian aligned
S.MUX.64.B.A	Store multiplex octlet big-endian aligned
S.MUX.64.L.A	Store multiplex octlet little-endian aligned

size				ordering		alignment
8						
16	32	64	128	L	B	
16	32	64	128	L	B	A

[46] S.8 need not specify byte ordering, nor need it specify alignment checking, as it stores a single byte.

Format

op ra,rb,rc



Description

A virtual address is computed from the sum of the contents of register ra and register rb. The contents of register rc, treated as the size specified, is stored in memory using the specified byte order.

If alignment is specified, the computed virtual address must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Definition

```
def Store(op,ra,rb,rc) as
    case op of
        S8,
        S16L, S16LA, S16B, S16BA,
        S32L, S32LA, S32B, S32BA,
        S64L, S64LA, S64B, S64BA,
        S128L, S128LA, S128B, S128BA:
            function ← NONE
        SAAS64BA, SAAS64LA:
            function ← AAS
        SCAS64BA, SCAS64LA:
            function ← CAS
        SMAS64BA, SMAS64LA:
            function ← MAS
        SMUX64BA, SMUX64LA:
            function ← MUX
    endcase
    case op of
        S8:
            size ← 8
        S16L, S16LA, S16B, S16BA:
            size ← 16
        S32L, S32LA, S32B, S32BA:
            size ← 32
        S64L, S64LA, S64B, S64BA,
        SAAS64BA, SAAS64LA:
            size ← 64
        SCAS64BA, SCAS64LA, SMAS64BA, SMAS64LA, SMUX64BA, SMUX64LA:
            size ← 64
        S128L, S128LA, S128B, S128BA:
            size ← 128
    endcase
    case op of
        S8,
        S16L, S16LA, S16B, S16BA,
        S32L, S32LA, S32B, S32BA,
```

```
        S64L, S64LA, S64B, S64BA,
        SAAS64BA, SAAS64LA:
            rsize ← 64
        SCAS64BA, SCAS64LA SMAS64BA, SMAS64LA, SMUX64BA, SMUX64LA:
            rsize ← 128
        S128L, S128LA, S128B, S128BA:
            rsize ← 128
    endcase
    case op of
        S8:
            align ← undefined
        S16L, S32L, S64L, S128L,
        S16B, S32B, S64B, S128B:
            align ← false
        S16LA, S32LA, S64LA, S128LA,
        S16BA, S32BA, S64BA, S128BA,
        SAAS64BA, SAAS64LA, SCAS64BA, SCAS64LA,
        SMAS64BA, SMAS64LA, SMUX64BA, SMUX64LA:
            align ← true
    endcase
    case op of
        S8:
            order ← undefined
        S16L, S32L, S64L, S128L,
        S16LA, S32LA, S64LA, S128LA,
        SAAS64LA, SCAS64LA, SMAS64LA, SMUX64LA:
            order ← L
        S16B, S32B, S64B, S128B,
        S16BA, S32BA, S64BA, S128BA,
        SAAS64BA, SCAS64BA, SMAS64BA, SMUX64BA:
            order ← B
    endcase
    a ← RegRead(ra, 64)
    b ← RegRead(rb, 64)
    VirtAddr ← a + b
    if align then
        if (VirtAddr and ((size/8)-1)) ≠ 0 then
            raise AccessDisallowedByVirtualAddress
        endif
    endif
    m ← RegRead(rc, rsize)
    case function of
        NONE:
            StoreMemory(VirtAddr,size,order,m_{size-1..0})
        AAS:
            c ← LoadMemory(VirtAddr,size,order)
            StoreMemory(VirtAddr,size,order,m_{63..0}+c)
            RegWrite(rc, 64, c)
        CAS:
            c ← LoadMemory(VirtAddr,size,order)
            if (c = m_{63..0}) then
                StoreMemory(VirtAddr,size,order,m_{127..64})
            endif
            RegWrite(rc, 64, c)
        MAS:
            c ← LoadMemory(VirtAddr,size,order)
            n ← (m_{127..64} & m_{63..0}) | (c & ~m_{63..0})
            StoreMemory(VirtAddr,size,order,n)
```

```
            RegWrite(rc, 64, c)
        MUX:
            c ← LoadMemory(VirtAddr,size,order)
            n ← (m127..64 & m63..0) | (c & ~m63..0)
            StoreMemory(VirtAddr,size,order,n)
    endcase
enddef
```

Exceptions

Reserved instruction
Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TLB
Access disallowed by local TLB
Access detail required by tag
Access detail required by local TLB
Access detail required by global TLB
Cache coherence intervention required by tag
Cache coherence intervention required by local TLB
Cache coherence intervention required by global TLB
Local TLB miss
Global TLB miss

E02

Store Immediate

These operations add the contents of a register to a sign-extended immediate value to produce a virtual address, and store the contents of a registerinto memory.

Operation codes

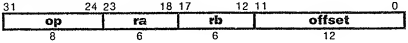
S.8.I[47]	Store byte immediate
S.16.B.A.I	Store double big-endian aligned immediate
S.16.B.I	Store double big-endian immediate
S.16.L.A.I	Store double little-endian aligned immediate
S.16.L.I	Store double little-endian immediate
S.32.B.A.I	Store quadlet big-endian aligned immediate
S.32.B.I	Store quadlet big-endian immediate
S.32.L.A.I	Store quadlet little-endian aligned immediate
S.32.L.I	Store quadlet little-endian immediate
S.64.B.A.I	Store octlet big-endian aligned immediate
S.64.B.I	Store octlet big-endian immediate
S.64.L.A.I	Store octlet little-endian aligned immediate
S.64.L.I	Store octlet little-endian immediate
S.128.B.A.I	Store hexlet big-endian aligned immediate
S.128.B.I	Store hexlet big-endian immediate
S.128.L.A.I	Store hexlet little-endian aligned immediate
S.128.L.I	Store hexlet little-endian immediate
S.AAS.64.B.A.I	Store add-and-swap octlet big-endian aligned immediate
S.AAS.64.L.A.I	Store add-and-swap octlet little-endian aligned immediate
S.CAS.64.B.A.I	Store compare-and-swap octlet big-endian aligned immediate
S.CAS.64.L.A.I	Store compare-and-swap octlet little-endian aligned immediate
S.MAS.64.B.A.I	Store multiplex-and-swap octlet big-endian aligned immediate
S.MAS.64.L.A.I	Store multiplex-and-swap octlet little-endian aligned immediate
S.MUX.64.B.A.I	Store multiplex octlet big-endian aligned immediate
S.MUX.64.L.A.I	Store multiplex octlet little-endian aligned immediate

size				ordering		alignment
8						
16	32	64	128	L	B	
16	32	64	128	L	B	A

[47] S.8.I need not specify byte ordering, nor need it specify alignment checking, as it stores a single byte.

Format

S.size.order.align.l ra,rb,offset



Description

A virtual address is computed from the sum of the contents of register ra and the sign-extended vale of the offset field. The contents of register rb, treated as the size specified, is stored in memory using the specified byte order.

If alignment is specified, the computed virtual address must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Definition

```
def StoreImmediate(op,ra,rb,offset) as
    case op of
        S8I,
        S16LI, S16LAI, S16BI, S16BAI,
        S32LI, S32LAI, S32BI, S32BAI,
        S64LI, S64LAI, S64BI, S64BAI,
        S128LI, S128LAI, S128BI, S128BAI:
            function ← NONE
        SAAS64BAI, SAAS64LAI:
            function ← AAS
        SCAS64BAI, SCAS64LAI:
            function ← CAS
        SMAS64BAI, SMAS64LAI
            function ← MAS
        SMUX64BAI, SMUX64LAI:
            function ← MUX
    endcase
    case op of
        S8I:
            size ← 8
        S16LI, S16LAI, S16BI, S16BAI:
            size ← 16
        S32LI, S32LAI, S32BI, S32BAI:
            size ← 32
        S64LI, S64LAI, S64BI, S64BAI, SAAS64BAI, SAAS64LAI,
        SCAS64BAI, SCAS64LAI, SMAS64BAI, SMAS64LAI, SMUX64BAI, SMUX64LAI:
            size ← 64
        S128LI, S128LAI, S128BI, S128BAI:
            size ← 128
    endcase
    case op of
        S8I,
        S16LI, S16LAI, S16BI, S16BAI,
        S32LI, S32LAI, S32BI, S32BAI,
        S64LI, S64LAI, S64BI, S64BAI,
        SAAS64BAI, SAAS64LAI:
```


E 04

```
            rsize ← 64
        S64AS64BAI, SCAS64LAI, SMAS64BAI, SMAS64LAI, SMUX64BAI, SMUX64LAI:
            rsize ← 128
        S128LI, S128LAI, S128BI, S128BAI:
            rsize ← 128
    endcase
    case op of
        S8I:
            align ← undefined
        S16LI, S32LI, S64LI, S128LI,
        S16BI, S32BI, S64BI, S128BI:
            align ← false
        S16LAI, S32LAI, S64LAI, S128LAI,
        S16BAI, S32BAI, S64BAI, S128BAI,
        SAAS64BAI, SAAS64LAI, SCAS64BAI, SCAS64LAI,
        SMAS64BAI, SMAS64LAI, SMUX64BAI, SMUX64LAI:
            align ← true
    endcase
    case op of
        S8I:
            order ← undefined
        S16LI, S32LI, S64LI, S128LI,
        S16LAI, S32LAI, S64LAI, S128LAI,
        SAAS64LAI, SCAS64LAI, SMAS64LAI, SMUX64LAI:
            order ← L
        S16BI, S32BI, S64BI, S128BI,
        S16BAI, S32BAI, S64BAI, S128BAI,
        SAAS64BAI, SCAS64BAI, SMAS64BAI, SMUX64BAI:
            order ← B
    endcase
    a ← RegRead(ra, 64)
    VirtAddr ← a + (offset11^50 || offset)
    if align then
        if (VirtAddr and ((size/8)-1)) ≠ 0 then
            raise AccessDisallowedByVirtualAddress
        endif
    endif
    m ← RegRead(rb, rsize)
    case function of
        NONE:
            StoreMemory(VirtAddr,size,order,m_size-1..0)
        AAS:
            b ← LoadMemory(VirtAddr,size,order)
            StoreMemory(VirtAddr,size,order,m_63..0+b)
            RegWrite(rb, 64, b)
        CAS:
            b ← LoadMemory(VirtAddr,size,order)
            if (b = m_63..0) then
                StoreMemory(VirtAddr,size,order,m_127..64)
            endif
            RegWrite(rb, 64, b)
        MAS:
            b ← LoadMemory(VirtAddr,size,order)
            n ← (m_127..64 & m_63..0) | (b & ~m_63..0)
            StoreMemory(VirtAddr,size,order,n)
            RegWrite(rb, 64, b)
        MUX:
```

```
            b ← LoadMemory(VirtAddr,size,order)
            n ← (m127..64 & m63..0) | (b & ~m63..0)
            StoreMemory(VirtAddr,size,order,n)
        endcase
enddef
```

Exceptions

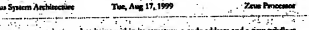
Reserved instruction
Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TLB
Access disallowed by local TLB
Access detail required by tag
Access detail required by local TLB
Access detail required by global TLB
Cache coherence intervention required by tag
Cache coherence intervention required by local TLB
Cache coherence intervention required by global TLB
Local TLB miss
Global TLB miss

EXHIBIT B

Second Supplemental Declaration of Korbin Van Dyke

Pages 19-20 and 123-130 of Zeus System Architecture manual
(from Microfiche Appendix of the '599 patent)

The gateway contains two data items within its structure, a code address and a new privilege level:



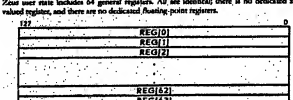
The virtual memory system can be used to designate a region of memory as containing gateways. Other data may be placed within the gateway region, provided that if an attempt is made to use the additional data as a gateway, that security cannot be violated. For example, 64-bit data or much pointers which are aligned to at least 4 bytes and are in little-endian byte order have $ptr[1:0]$, so that the privilege level cannot be raised by attempting to use the additional data as a gateway.

User State

The user state consists of hardware data structures that are accessible to all conventional compiled code. The Zeus user state is designed to be as regular as possible, and consists only of the general registers, the program counter, and virtual memory. There are no specialized registers for condition codes, operating modes, rounding modes, integer multiply/divide, or floating point values.

General Registers

Zeus user state includes 64 general registers. All are identical; there is no dedicated zero-valued register, and there are no dedicated floating-point registers.



Some Zeus instructions have 64-bit register operands. These operands are sign-extended to 128 bits which written to the register file, and the low-order 64 bits are chosen when read from the register file.

Definition

```
def val ← RegRead(rn, size)
    case size of
        64:
            val ← REG[rn]63:0
        128:
```



```
            val ← REG[rn]
        endcase
endcase

def RegWrite(rn, size, val)
    case size of
        64:
            REG[rn] ← val63:0 || val63:0
        128:
            REG[rn] ← val127:0
    endcase
endcase
```

Program Counter

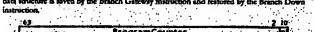
The program counter contains the address of the currently executing instruction. This register is implicitly manipulated by branch instructions, and read by branch instructions that save a return address in a general register.

Privilege Level

The privilege level register contains the privilege level of the currently executing instruction. This register is implicitly manipulated by branch gateway and branch down instructions, and read by branch gateway instructions that save a return address in a general register.

Program Counter and Privilege Level

The program counter and privilege level may be packed into a single octet. This combined data structure is saved by the Branch Gateway instruction and restored by the Branch Down instruction.



System state

The system state consists of the facilities not normally used by conventional compiled code. These facilities provide mechanisms to execute such code in a fully virtual environment. All system state is memory mapped, so that it can be manipulated by compiled code.

Z-ce System Architecture Thu, Sep 12, 1999 Instruction Set

Store

There opcodes add the contents of two registers to produce a virtual address, and store the contents of a register into memory.

Assembler codes	
ST.B	Store byte
ST.B.AS	Store byte, aligned address
ST.B.AI	Store byte, aligned instruction
ST.B.AD	Store byte, aligned data
ST.S	Store quarter (16 bits)
ST.S.AS	Store quarter, aligned address
ST.S.AI	Store quarter, aligned instruction
ST.S.AD	Store quarter, aligned data
ST.W	Store octet (32 bits)
ST.W.AS	Store octet, aligned address
ST.W.AI	Store octet, aligned instruction
ST.W.AD	Store octet, aligned data
ST.T	Store hextet (64 bits)
ST.T.AS	Store hextet, aligned address
ST.T.AI	Store hextet, aligned instruction
ST.T.AD	Store hextet, aligned data
ST.MEX.64	Store multiple octets
ST.MEX.64.AL	Store multiple octets, aligned address

Selection							
[number] [name]		op	size		alignment		opcode
byte			8				B
octet			32	64	128		C
hextet	aligned		64	128		A	D
multiples		MEX					E

Format

op rs2,rs1,rd

Pick one set specify type address, and don't specify operand checking, or enter a single byte.

123 Monolithy

Z-ce System Architecture Thu, Sep 12, 1999 Instruction Set

Description

An opcode also commands to form, is specified by the instruction. A virtual address is computed from the sum of the contents of register rs1 and the contents of register rs2. The contents of register rd is stored at the computed virtual address.

If the address specified by the computed virtual address is not aligned on a boundary, an exception is raised.

Assembler

```
def StoreOpCode rs
    rs2 = rs
    rs1 = 0
    rd = 0
    $stdout.print "#{self.class.name} #{rs}\n"
    $stdout.print "  rs2 = #{rs2}\n"
    $stdout.print "  rs1 = #{rs1}\n"
    $stdout.print "  rd = #{rd}\n"
    $stdout.print "  opcode = #{opcode}\n"
    $stdout.print "  machine code = #{machine_code}\n"
    $stdout.print "  machine code = #{machine_code.to_s(2)}\n"
    $stdout.print "  machine code = #{machine_code.to_s(16)}\n"
    $stdout.print "  machine code = #{machine_code.to_s(16).upcase}\n"
    machine_code = machine_code << 6
    machine_code = machine_code | rs2
    machine_code = machine_code << 6
    machine_code = machine_code | rs1
    machine_code = machine_code << 6
    machine_code = machine_code | rd
    machine_code = machine_code << 6
    machine_code = machine_code | opcode
    $stdout.print "  machine code = #{machine_code}\n"
    $stdout.print "  machine code = #{machine_code.to_s(2)}\n"
    $stdout.print "  machine code = #{machine_code.to_s(16)}\n"
    $stdout.print "  machine code = #{machine_code.to_s(16).upcase}\n"
    $stdout.print "  machine code = #{machine_code.to_s(16).upcase.rjust(8,'0')}\n"
    $stdout.print "  machine code = #{machine_code.to_s(16).upcase.rjust(8,'0').scan(/../).join(' ')}\n"
    $stdout.print "  machine code = #{machine_code.to_s(16).upcase.rjust(8,'0').scan(/../).reverse.
```

...store

...
S16L, S16AL, S16R, S16AR,
S32L, S32AL, S32R, S32AR,
S64L, S64AL, S64R, S64AR,
S128L, S128AL, S128R, S128AR:
StoreMemory(V, VirtAddr,size,order,data<size-1..0>)
SADDL64AB, SADDR64AB:
lock
a <- LoadMemory(V, VirtAddr,size,order)
m <- if(27..64 = 0x0 of 2 | 4 : -0x3..0)
StoreMemory(V, VirtAddr,size,order,m)
endlock
endcase
endif

Exceptions

Access disallowed by virtual address
Access disallowed by seg
Access disallowed by global TB
Access disallowed by local TB
Access denied requested by seg
Access denied requested by local TB
Access denied requested by global TB
Local TB miss
Global TB miss

Store Double Compare Swap

These operations compare two 64-bit values in registers against two 64-bit values read from two 64-bit memory locations, as specified by two effective addresses in a register, and if equal, store two new 64-bit values from a register into the memory locations. The values read from memory are transferred and placed in a register.

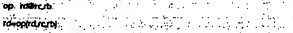
Operation codes

SDCS64AB	Store double compare swap octlet aligned big-endian
SDCS64AL	Store double compare swap octlet aligned little-endian

Format

op rd,rc,rb

rd=oppd,rc,rb



Description

Two virtual addresses are extracted from the low order bits of the contents of registers rc and rb. Two 64-bit comparison values are extracted from the high order bits of the contents of registers rc and rb. Two 64-bit replacement values are extracted from the contents of register rd. The contents of memory using the specified byte order are read from the specified addresses, treated as 64-bit values, compared against the specified comparison values, and if both read values are equal to the comparison values, the two replacement values are written to memory using the specified byte order. If either are unequal, no values are written to memory. The loaded values are extracted and placed in the register specified by rd.

The virtual addresses must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Definition

```
def StoreDoubleCompareSwap(op,rd,rc,rb) as
    size <- 64
    base <- bigend
    case op of
        SDCS64AL:
            order <- 1
        SDCS64AB:
```

```
            order <- 3
        endcase
        c <- RegRead(rc, 128)
        b <- RegRead(rb, 128)
        d <- RegRead(rd, 128)
        if PSR.AE[0] or PSR.AE[3] then
            raise AccessDisallowedByVirtualAddress
        endif
        lock
            a <- LoadMemory(PSR.ASI, a*64, 64, order) || LoadMemory(PSR.ASI, a*64+8, 64, order)
            if a[127:64] != c[127:64] or a != c then
                StoreMemory(PSR.ASI, a*64, 64, order, d[127:64]
                StoreMemory(PSR.ASI, a*64+8, 64, order, d[63:0]
            endif
        endlock
        RegWrite(rd, 128, a)
    endif
endcase
```

Exceptions

Access disallowed by virtual address
Access disallowed by reg
Access disallowed by global TB
Access disallowed by local TB
Access denied required by reg
Access denied required by local TB
Access denied required by global TB
Local TB miss
Global TB miss

Store Immediate

These operations add the contents of a register to a sign-extended immediate value to produce a virtual address, and store the contents of a register into memory.

Operation codes

ST8,I	Store immediate byte
ST16,AB	Store immediate double aligned big-endian
ST16,B	Store immediate double big-endian
ST16,AL	Store immediate double aligned little-endian
ST16,L	Store immediate double little-endian
ST32,AB	Store immediate quadlet aligned big-endian
ST32,B	Store immediate quadlet big-endian
ST32,AL	Store immediate quadlet aligned little-endian
ST32,L	Store immediate quadlet little-endian
ST64,AB	Store immediate octlet aligned big-endian
ST64,B	Store immediate octlet big-endian
ST64,AL	Store immediate octlet aligned little-endian
ST64,L	Store immediate octlet little-endian
ST128,AB	Store immediate hextet aligned big-endian
ST128,B	Store immediate hextet big-endian
ST128,AL	Store immediate hextet aligned little-endian
ST128,L	Store immediate hextet little-endian
STMOD64,AB	Store multiple immediate octet aligned big-endian
STMOD64,AL	Store multiple immediate octet aligned little-endian

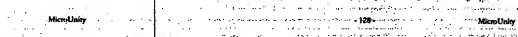
Selection

number format	op	size	alignment	ordering
byte		8		
integer		16 32 64 128		U 8
integer aligned		16 32 64 128	A	L B
multiple	MUX	64	A	L B

Format

S.op.Size.align.order rd,rc,offset

sopsizealignorder(rd,rc,offset)



[1] It need not specify byte ordering, nor need it specify alignment checking, as it stores a single byte.